SOFT WARE APPLICATION LIFE CYCLE AND MANAGEMENT FOR BROADCAST APPLICATIONS

BACKGROUND OF THE INVENTION

This application claims the benefit of U.S. Provisional Application No. 60/103,943, filed October 13, 1998.

The present invention provides a software architecture for managing applications at a television set-top terminal.

The following acronyms and terms are used:

API - Application Programming Interface;

ATSC - Advanced Television Systems Committee;

DASE - ATSC T3/S17 Digital TV Application Software Environment;

DAVIC - Digital Audio-Visual Council;

DTV - Digital Television;

EPG - Electronic Program Guide;

IRD - Integrated Receiver Decoder;

ISO - International Standards Organization;

JVM - Java Virtual Machine;

PSIP - Program and System Information Protocol (for Terrestrial Broadcast and Cable);

RAM - Random Access Memory; and

UML - Unified Modeling Language.

A set-top terminal, also referred to as an IRD or a subscriber terminal, is a device that receives and decodes television signals for presentation by a television. The signals can be delivered over a satellite, through a cable plant, or by means of terrestrial broadcast, for example. Various applications have been proposed, or are currently available, via modern set tops, including video on demand (VOD), audio on demand, pay-per-view, interactive shopping, electronic commerce, electronic program guides, Internet browsers, mail services (e.g., text e-mail, voice mail, audio mail, and/or video mail), telephony services, stock ticker, weather data, travel information, games, gambling, banking, shopping, voting, and others. Applications may also enable Internet connectivity and possibly Internet-based telephony. The set top functionality is enabled through specialized hardware and software.

Moreover, with the increasing integration of computer networks such as the Internet, telephony networks, and broadband distribution networks, many opportunities arise for providing new types of applications.

The applications may be communicated to a set-top terminal via a network, loaded locally (e.g., via a smart card), or installed at the time of manufacture, for example.

In particular, the DASE Application Management System Service requirements propose a number of requirements for managing the applications at a set-top terminal. This is a section from the ATSC T3/S17 draft specification which describes application management related requirements (Section 13).

Accordingly, it would be desirable to provide set-top software for managing applications at a set-top terminal. The software should provide an API for retrieving and registering new applications that are received at the terminal, e.g., via a download from a headend, and providing an identifier for each application.

The API should be independent of the operating system and hardware of the terminal.

It would be desirable to provide an ITU-T X.731-based mechanism for application monitoring and control.

The mechanism should control starting, stopping, pausing and resuming of the applications.

The mechanism should enable an application to advertise its state to other applications, and allow the other applications to access the advertised state.

The mechanism should allow retrieving of application and resource version information.

The mechanism should allow accessing of application location information.

The mechanism should provide verification of the integrity of an application, and validation of the suitability of the application for use in the set-top terminal.

The mechanism should notify the user of the existence of a new application after it is registered.

The mechanism should provide an administrative locking or unlocking of an application.

The mechanism should advertise the operational state. alarm status, and availability status of an application.

The architecture should be compatible with Java(tm), ActiveX(tm) or an equivalent type of component based object-oriented technology.

The mechanism should be suitable for use with any application at a terminal, regardless of how it was received or installed.

The present invention provides a system having the above and other advantages.

SUMMARY OF THE INVENTION

The present invention provides a software architecture for managing applications in a set-top television terminal.

A television set-top terminal includes a computer readable medium having computer program code means, and means for executing the computer program code means to implement an Application Programming Interface (API). With the API, application data which defines applications is recovered at the terminal according to locators associated with the applications. For example, the locator may be a PID, channel number, channel name, Transport Stream ID, Service ID, a combination thereof, or something else. The locator may be in the form of a Uniform Resource Locator (URL).

The applications are registered and installed at the terminal, and a user is notified of the presence of the applications after registration and installation thereof. Thus, the user is notified when the application is made available locally at the terminal and is ready to be invoked/started.

An application may use a resource, usually a device, function or a process on the receiver (e.g. tuner, modem, database, etc.)

The API enables the retrieval of the applications as downloadable software applications, or broadcast software applications.

The API may be independent of an operating system and hardware of the terminal.

The API may provide an ITU-T X.731-based mechanism for monitoring and controlling the applications.

The API may enable running and subsequent stopping of the applications.

The API may enable pausing of the applications once they are running, and subsequent resuming of the applications.

The API may enable particular ones of the applications to advertise their respective states to other applications.

The API may enable at least one of the other applications to access the advertised state of at least one of the particular advertising applications. An application state may have several different values (enabled, disabled, etc.) To access a state means to have the ability to learn about the current state value.

The API may enable retrieval of version information associated with the applications.

The API may enable verification of the integrity of the applications. Integrity in this case may mean that the code that was received by the receiver is legal and valid based on the programming language specification used to code the application (e.g., Java programming language, etc.)

The API may enable validation of the suitability of the applications for the terminal.

The API may enable administrative locking and unlocking of the applications.

The API may enable particular ones of the applications to advertise respective alarm statuses, availability statuses, procedural statuses, operational states, administrative states and usage states thereof to other ones of the applications.

A corresponding method is also presented.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1 shows package relationships and dependencies in accordance with the present invention.

FIG. 2 represents application-related classes and interfaces and their relationships in accordance with the present invention.

FIG. 3 describes those classes and interfaces related to state management in accordance with the present invention.

FIG. 4 depicts relationships between the utility classes and interfaces in accordance with the present invention.

FIG. 5 is an interaction/sequence diagram showing how an EPG application, which displays video as well as data channels with applications available on them to the user, can invoke a download and subsequent execution of the downloaded application in accordance with the present invention.

FIG. 6 shows a set of interactions/sequences that demonstrates how one application can be managed by an application manager and monitored by an agent in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The present invention provides a software architecture for managing applications in a set-top television terminal.

1. Overview

The present invention specifies an API which satisfies the DASE Application Management System Service requirements.

Note that portions of the disclosure were generated automatically from Rational Rose(tm) CASE tool, developed by Rational Software Corporation, USA. Exceptions and pre- and post-conditions associated with individual methods are not shown. Exceptions will be shown in a Javadoc format.

The figures use the Rational Rose (tm) depiction of the UML. FIGs 1-4 are class diagrams and FIGs 5 and 6 are sequence (or interaction) diagrams. A class diagram represents the static structure of a system, and shows a pattern of behaviors that the system exhibits. This is accomplished by showing the existence of classes and their relationships. Each class is represented by a box with three sections. The top section lists the class name. The middle section denotes a list of attributes, and the bottom section denotes a list of operations.

A solid or dashed line between classes denotes an association or dependency. A white diamond tip denotes aggregation by reference, while a black diamond tip denotes aggregation by value. A triangular arrowhead denotes a restricted navigation, e.g., inheritance of operation but not of structure.

A class is a template that defines a data structure, method and function calls for an object. An interface defines a set of methods/function calls that can be manipulated by a class. The class provides the code for implementing an interface.

2. REQUIREMENTS

The proposed API addresses the following requirements:

1. The API will provide a mechanism to retrieve a downloadable or broadcast application.

This is done via a registration mechanism, which can specify a URL of an application (obtained from the PSIP API or its extensions based on T3/S13 and S16 work) to be downloaded and made available. ATSC T3/S13 and T3/S16 specifications define protocols for delivering applications to the receiver, signaling their presence in the transport stream and providing information about the applications. The URL is used as an identifier for applications in an example embodiment, but other identifiers may be used.

2. The API will provide a mechanism to install and uninstall an application.

The Registration mechanism installs the application so that it can be invoked/started.

3. The API will provide a mechanism to initialize (launch), start, and stop an application.

The Application interface provides methods to perform these actions.

4. The API will provide a mechanism to pause and resume an application.

The Application interface provides methods to perform these actions.

5. The API will provide a mechanism for applications to maintain an access state.

Each application which implements the ObjectStates interface can be managed in a standard way defined by ITU-T. ITU-T X.731 is an international standard which defines management states, status codes and state transitions for manageable objects (devices, resources, applications, etc.) The API will provide a mechanism to retrieve version information for the application and its resources, including required APIs.

The ApplicationInformation interface allows the retrieval of the above information.

6. The API will provide a mechanism to access application location information.

The application location information can be represented in a URL format (to be standardized by ATSC) in the DAVIC Locator class. A locator is an opaque object which encapsulates a URI (Universal Resource Identifier) of a particular resource (an application in this case).

7. The API will provide a mechanism to verify an application's integrity and validate its correctness. For example, this may mean that it does not include viruses or will not do any damage to the receiver.

The JVM verifier satisfies this requirement; therefore, it is not necessary define a specific API to do so.

8. The API will provide a registration mechanism allowing the application to notify the user of its existence.

This is done in conjunction with the PSIP and S13 APIs, which provide information about a specific application. This information can be used to download an application by registering it with the ApplicationRegistry. Once registered, a user can use it.

3. DESCRIPTION

This proposal consists of two main packages: org.atsc.application and org.atsc.management, and a helper package org.atsc.utility. The first package includes application-specific classes and interfaces. The other package represents classes and interfaces that are related to managing application states based on the ITU-T management standard. The latter is separated into its own package since it can be applied to any manageable object such as DTV receiver resources, or downloadable applications.

An application is free to support a subset of the defined states and status attributes as appropriate to the specific application. DASE may mandate a subset of these in order to provide for a better interoperability between applications with respect to management. Some applications may be very simple, and some of the states and statuses defined by the X.731 standard may not be applicable. ATSC may define a minimal set of states and status codes that are supported by all applications. Some, more complex, applications may support more. For example, some applications may not support the "degraded" Availability Status - they either work or they don't - nothing in between.

4. OBJECT MODEL

FIG. 1 shows the package relationships and dependencies in accordance with the present invention. The org.atsc.application package 105 uses classes and interfaces defined in the org.atsc.management package 110, org.atsc.utility package 115 and org.davic.net package 120. The packages are logically related by the dashed arrows, which denote dependency.

FIG. 2 represents application-related classes and interfaces and their relationships in accordance with the present invention. An interface is labeled by <<interface>>, while those not so labeled are classes. The classes include: RegistryFactory 215, Exception 220, ApplicationAvailabilityException 225, ApplicationAlreadyRegisteredException 230, ApplicationNotRegisteredException 235, ApplicationRegistryEvent 245, and EventObject 250.

The interfaces include: ApplicationRegistry 205, Registry 210, ApplicationCause 240, ApplicationRegistryListener 255, StateChangeListener 260, ObjectStates 265, Application 270, and ApplicationInformation 275.

FIG. 3 describes those classes and interfaces related to state management in accordance with the present invention. Like-numbered elements correspond to one another in the figures.

The classes and interfaces include: AdministrativeState 305, OperationalState 310, UsageState 315, ObjectState 320, AlarmStatus 325, AvailabilityStatus 330, ProceduralStatus 335, ResourceStateException 340, SourceIndicator 345, and StateChangeEvent 350.

FIG. 4 depicts relationships between the utility classes and interfaces in accordance with the present invention. The classes and interfaces include RegistryType 405.

5. INTERACTION DIAGRAMS

The following sections will describe example interactions between the application-related classes, and show how other objects can use the Application Management API. Since an application is built from objects, an API accessed by other objects means that the API is accessed by parts (objects) of other applications, or by code present on the terminal.

5.1 Application Registration

FIG. 5 is an interaction/sequence diagram showing how an EPG application, which displays video as well as data channels with applications available on them to the user, can invoke a download and subsequent execution of the downloaded application in accordance with the present invention. The diagram was generated using Rational Rose(tm) software.

A number of example objects are provided, including "user" 505, "EPG: Application" 270' (an example of the Application interface 270 of FIG. 2), "PSIP Database" 515, "dataChannel" 520, "factory:Registry Factory" 215, "registry:Application Registry" 205, "downloader" 525, and "app:Application" 270'' (an example of the Application interface 270 of FIG. 2).

The EPG retrieves the application information including the URL (Locator), gets access to the ApplicationRegistry via the RegistryFactory and requests a registration of the new application.

When an application is registered with the Application Registry, the application locator (URL) is used to download the application. When it is available, the registry fires (e.g., sends/emits) an event to all listeners to indicate that the new application is registered and available. The EPG application listens to these events and notifies the user of the new application availability.

Once the application is downloaded and installed, the user can request its execution via the registry. The registry actually starts the application.

The above sequence may be implemented via the following example steps 1-13:

1. The "EPG:Application" object 270' calls (e.g., invokes) the "getVirtualChannels" method from the "PSIP Database" object 515;

2. The "EPG:Application" object 270' calls the "getDataApps" method from the "dataChannel" object 520;

3. The "EPG:Application" object 270' calls the "displayApps" method from the "user" object 505;

4. The "user" object 505 calls the "selectApp" method from the "EPG:Application" object 270';

5. The "EPG:Application" object 270' calls the "getRegistry(String)" method from the "factory:Registry Factory" object 215;

6. The "EPG:Application" object 270' calls the "registerApplication(Locator)" method from the "registry:Application Registry" object 205;

7. The "registry:Application Registry" object 205 calls the "download" method from the "downloader" object 525;

8. The "registry:Application Registry" object 205 calls the "registryChange(ApplicationRegistryEvent)" method from the "EPG:Application" object 270';

9. The "EPG:Application" object 270' calls the "getApplicationInformation(Locator)" method from the "registry:Application Registry" object 205;

10. The "EPG:Application" object 270' calls the "displayAppinfo" method from the "user" object 505;

11. The "user" object 505 calls the "invokeApp" method from the "EPG:Application" object 270';

12. The "user" object 505 calls the "startApplication(Locator)" method from the "registry:Application Registry" object 205; and

13. The "registry:Application Registry" object 205 calls the "start()" method from the "app:Application" object 270''.

5.2 Managing Application States

FIG. 6 shows a set of interactions/sequences that demonstrates how one application can be managed by an application manager and monitored by an agent in accordance with the present invention.

A number of example objects are provided, including "app1:application" 270'', "appManager:StateChange" 260', "event:StateChangeEvent" 350', and "agent:StateChangeListener" 260'' (or 270'').

The agent registers as a StateChangeListener to a specific application. The application changes its internal states based on internal or external causes. In this example, the application was suspended, which changed its OperationalState to DISABLED. The application creates a StateChangeEvent and sends it to all registered listeners; in this case the agent. The agent can determine the state that changed and its old and new values by interrogating the event, e.g., by calling the methods available on the StateChangeEvent object, such as getOldValue(), getnewValue(), etc.

The above sequence may be implemented via the following example steps 1-11:

1. The "appManager:StateChange" object 260' calls the "start()" method from the "app1:Application" object 270'';

2. The "agent:StateChangeListener" object 260'' calls the "addStateChangeListener" method from the "app1:Application" object 270'';

3. The "appManager:StateChange" object 260' calls the "suspend()" method from the "app1:Application" object 270'';

4. The "app1:Application" object 270'' calls the "StateChangeEvent" method from the "event:StateChangeEvent" object 350';

5. The "app1:Application" object 270'' calls the "stateChange(StateChangeEvent)" method from the "agent:StateChangeListener" object 260'';

6. The "agent:StateChangeListener" object 260'' calls the "getState()" method from the "event:StateChangeEvent" object 350';

7. The "agent:StateChangeListener" object 260'' calls the "getOldValue()" method from the "event:StateChangeEvent" object 350';

8. The "agent:StateChangeListener" object 260'' calls the "getNewValue()" method from the "event:StateChangeEvent" object 350';

9. The "appManager:StateChange" object 260' calls the "resume()" method from the "app1:Application" object 270'';

10. The "app1:Application" object 270'' calls the "StateChangeEvent" method from the "event:StateChangeEvent" object 350'; and

11. The "app1:Application" object 270'' calls the "stateChange(StateChangeEvent)" method from the "agent:StateChangeListener" object 260''.

6. Class And Interface Description

As many APIs as possible are defined as interfaces rather than as classes. This provides more freedom and fewer restrictions for the API implementation. Since Java interfaces don't have constructors or static methods, some interfaces such as the ApplicationRegistry have an associated RegistryFactory class that returns the appropriate implementation of the ApplicationRegistry interface. The RegistryFactory class may be based on the Factory Method pattern, which, as is known from the field of object-oriented programming, is a methodology and structure for solving a problem.

Section 6.1 describes the application-related packages.

6.1 org.atsc.application

This package includes classes and interfaces related to application lifecycle, registration and management.

6.1.1 Application

This class represents a base class of all downloadable applications. It provides a basic application lifecycle support and additional descriptive information about the application in the form of an ApplicationInfo class.

This class implements the GenericStates interface to add management capability to a downloadable application. This interface provides a uniform mechanism to manage any object in a standard way. An Application can support a subset of these states as appropriate to the specific application.

The class is derived from ObjectStates.

Public Operations:

start () :

Called by the controlling process to initiate an execution of an application. The application can acquire any needed resources, perform its initialization and start execution.

If this application supports the Administrative State, it will throw an exception when it is in the Locked state.

Public operations are those methods that may be called and used by other objects since they are visible outside of the object (e.g., class). In contrast, private operations are visible only to the class itself.

stop () :

Called by the controlling process to stop an execution of this application. The application should release all resources and terminate.

suspend () :

Called by the controlling process to temporarily pause an execution of this application. The application is not required to give up its resources unless it is requested to do so using a different mechanism.

If this application supports the Operational State, it will change the state to Disabled upon completion of this method.

resume () :

Called by the controlling process to resume an execution of the application that was previously suspended.

If this application supports the Operational State, it will change the state to Enabled upon completion of this method.

getApplicationID () : Locator

Called to determine application identification represented as a Locator such as a URL.

6.1.2 ApplicationInformation

This class provides additional information about an Application, such as a name, version number, author, etc.

Public Operations:

getTitle () : String

Returns a short description of the application, such as its name or title.

getVendor () : String

Returns the name of the application vendor or author.

getVersion () : String

Returns the version of this implementation. It consists of a string assigned by the vendor of this implementation.

Version numbers use a "Dewey Decimal" syntax that consists of positive decimal integers separated by periods

".", for example, "2.0" or "1.2.3.4.5.6.7". This allows an extensible number to be used to represent major, minor, micro, etc. versions. The version number must begin with a number.

getRequiredProfile () : String

Returns the minimum profile identifier, such as DASE profile ID, that is expected by this application to run.

getSource () : Locator

Returns the original source of the application in a URL format. The source is where the application came from (e.g., channel 39, HBO, CNBC, etc.)

6.1.3 ApplicationRegistry

This interface provides a limited access to the Application Registry. It allows other applications to get information about existing applications, to show an interest in a particular application (registering it) and getting access to applications themselves. The interface is derived from Registry.

Public Operations:

registerApplication (applicationID : Locator) :

Called to add this application from the registry. The application is specified via its

Locator (URL). The registry is responsible for locating the application, downloading it and notifying the caller of its availability.

This is a non-blocking method; it will return immediately after checking the request. The ApplicationAvailableEvent will be sent to all ApplicationRegistryListeners with an indication of the result of registering this application.

deregisterApplication (applicationID:Locator):

Called to remove this application from the registry.

getApplicationInformation (applicationID:Locator) : ApplicationInformation

Called to obtain a description of this application. The application is identified by a Locator (URL).

getApplication (applicationID:Locator) : Application

Called to get access to a specified loaded and installed application.

This method is protected via the security mechanisms in order to protect unauthorized access to applications.

getApplications() : Application[]

This method allows a retrieval of all registered applications. This method is protected via the security mechanisms in order to protect unauthorized access to applications.

startApplication(applicationID:Locator):

Called to invoke a previously registered application. The method call returns as soon as the requested application starts executing in its own thread space.

This method is protected via the security mechanisms in order to protect unauthorized access to applications.

addApplicationRegistryListener (listener: ApplicationRegistryListener) :

Called to register for events generated by the ApplicationRegistry.

removeApplicationRegistryListener (listener : ApplicationRegistryListener) :

Called to deregister for events generated by the ApplicationRegistry.

6.1.4 ApplicationRegistryListener

This interface allows an object to listen to changes made to the ApplicationRegistry.

Public Operations:

registryChange() : ApplicationRegistryEvent

This method of all registered ApplicationRegistryListeners is called by the ApplicationRegistry object when an ApplicationRegistryEvent is fired.

6.1.5 ApplicationRegistryEvent

Derived from EventObject.

Public Operations:

getApplicationInformation() : ApplicationInformation

Called to determine which application caused the event.

getCause():short

Called to determine what caused this event.

6.1.6 ApplicationAvailabilityException

This exception is thrown when the requested application availability condition was violated. It is derived from Exception.

6.1.7 ApplicationNotRegisteredException

Derived from ApplicationAvailabilityException

6.1.8 ApplicationAlreadyRegisteredException

Derived from ApplicationAvailabilityException

6.1.9 ApplicationCause

Public Attributes:

REGISTERED : short = 1

Application was registered in the registry. "Short" is one format for integers (2 bytes vs. 4 bytes).

DEREGISTERED : short = 2

Application was deregistered from the registry.

STARTED : short = 3

Application was started.

6.2 org.atsc.management

This package includes classes and interfaces related to object management. It can be applied in its entirety or as a subset as relevant to the specific managed entity. It is applicable for managing state and status attributes of DTV receiver resources as well as applications.

It is based on the ITU-T X.731 standard for State Management.

6.2.1 AdministrativeState

An interface that defines Masks for different Administrative States:

-locked: The resource is administratively prohibited from performing services for its users. This could relate to a local lockout, such as a parental lockout of certain channels or applications, but can also be used to remotely (from the headend, uplink or cable operator) "lock" the application so that the user cannot start it, e.g., if a problem with an application is detected.

-Unlocked: The resource is administratively permitted to perform the services to users. This is independent of its inherent operability.

-Shutting down: Use of resource is administratively permitted to the existing instances of users only. The manager may at any time cause the object to revert to Unlocked state.

Public Attributes:

UNLOCKED : int = 0x00000001

LOCKED : int = 0x00000002

SHUTTING_DOWN : int = 0x00000004

ADMIN_TYPE : short = 1

Public Operations:

getAdministrativeState () : int

Called to get the current value of the Administrative State.

setLock (administrativeState : int) :

Called to change the value of the Administrative State.

6.2.2 OperationalState

An interface that defines the Operational state for Resources and Applications:

-Disabled: The resource is totally inoperable and unable to provide the service to the users.

-Enabled: The resource is partially operable and available for use.

Public Attributes:

DISABLED : int = 0x8

ENABLED : int = 0x10

OPERATIONAL_TYPE : short = 2

Public Operations:

getOperationalState () : int

Called to get the current value of the Operational State.

6.2.3 AlarmStatus

Interface that defines all the alarm states. When the value of this attribute is an empty set, this implies that none of the status conditions described below are present:

-under repair: The resource is currently being repaired. When the under repair value is present, the operational state is either disabled or enabled.

-critical: One or more critical alarms indicating a fault have been detected in the resource, and have not been cleared. The operational state of the managed object can be disabled or enabled.

-major: One or more major alarms indicating a fault have been detected in the resource, and have not yet been cleared. The operational state of the managed object can be disabled or enabled.

-minor: One or more minor alarms indicating a fault have been detected in the resource, and have not yet been cleared. The operational state of the managed object can be disabled or enabled.

-alarm outstanding: One or more alarms have been detected in the resource. The condition may or may not be disabling. If the operational state is enabled, additional attributes, particular to the managed object class, may indicate the nature and cause of the condition and the services that are affected.

The presence of the above alarm state conditions do not suppress the generation of future fault related notifications.

Public Attributes:

UNDER_REPAIR : int = 0x00000001

CRITICAL : int = 0x00000002

MAJOR : int = 0x00000004

MINOR : int = 0x00000008

ALARM_OUTSTANDING : int = 0x0010

ALARM_TYPE : short = 8

Public Operations:

clearAlarm (alarm : int) :

Called to clear a specific alarm. The controlling process has acted on the alarm.

getAlarmStatus () : int

Called to get the current set of values of the Alarm Status.

6.2.4 AvailabilityStatus

Defines the Availability status. When the value of this attribute is an empty set, this implies that none of the status conditions described below are present:

-in test: The resource is undergoing a test procedure. If the administrative state is locked or shutting down, then normal users are precluded from using the resource and the control status attribute has the value reserved for test. Tests that do not exclude additional users can be present in any operational or administrative state but the reserved for test condition should not be present.

-failed: The resource has an internal fault that prevents it from operating. The operational state is disabled.

-power off: The resource requires power to be applied and is not powered on. For example, a fuse or other protection device is known to have removed power, or a low voltage condition has been detected. The operational state is disabled.

-off line: The resource requires a routine operation to be performed to place it online and make it available for use. The operation may be manual or automatic, or both. The operational state is disabled.

-off duty: The resource has been made inactive by an internal control process in accordance with a predetermined time schedule. Under normal conditions the control process can be expected to reactivate the resource at some scheduled time, and it is therefore considered to be optional. The operational state is enabled or disabled.

-dependency: The resource cannot operate because some other resource on which it depends (e.g., a resource not represented by the same managed object) is unavailable. For example, a device is not accessible because its controller is powered off. The operational state is disabled.

-degraded: The service available from the resource is degraded in some respect, such as in speed or operating capacity. Failure of a test or an unacceptable performance measurement has established that some or all services are not functional or are degraded due to the presence of a defect. However, the resource remains available for service, either because some services are satisfactory or because degraded service is preferable to no service at all. Object-specific attributes may be defined to represent further information indicating, for example, which services are not functional and the nature of the degradation. The operational state is enabled.

-not installed: The resource represented by the managed object is not present, or is incomplete.

For example, a plug-in module is missing, a cable is disconnected or a software module is not loaded. The operational state is disabled.

-log full: This indicates a log full condition, the semantics of which are defined in CCITT Rec. X.735 | ISO/IEC 10164-6.

Public Attributes:

INTEST : int = 0x00000400
FAILED : int = 0x00000800
POWEROFF : int = 0x00001000
OFFLINE : int = 0x00002000
OFFDUTY : int = 0x00004000
DEPENDENCY : int = 0x00008000
DEGRADED : int = 0x00010000
NOT_INSTALLED : int = 0x00020000
LOG_FULL : int = 0x00040000
AVAILABILITY_TYPE : short = 32

Public Operations:

getAvailabilityStatus () : int

Called to get the current set of values of the Availability Status.

6.2.5 ProceduralStatus

An interface that defines the Procedural status.

The procedural status attribute is supported only by those classes of managed objects that represent some procedure (e.g., a test process) which progresses through a sequence of phases. Depending upon the managed object class definition, the procedure may be required to reach certain phase for the resource to be operational and available for use (i.e., for the managed object to be enabled). Not all phases may be applicable to every class of managed object. If the value of this attribute is an empty set, the managed object is ready, for example, the initialization is complete.

When the value of this attribute is an empty set, this implies that none of the status conditions described below are present.

-initialization required: The resource requires initialization to be invoked by the manager before it can perform its normal functions, and this procedure has not been initiated. The manager may be able to invoke such initialization through an action. The terminating condition may also be present. The operational state is disabled.

-not initialized: The resource requires initialization before it can perform its normal functions, and this procedure has not been initiated. The resource initializes itself autonomously, but the operational state may be either disabled or enabled, depending upon the managed object class definition.

-initializing: The resource requires initialization before it can perform its normal functions, and this procedure has been initiated but is not yet complete. When this condition is present, the initialization required condition is absent, since initialization has already begun. The operational state may be disabled or enabled, depending upon the managed object class definition.

-reporting: The resource has completed some processing operation and is notifying the results of the operation, e.g., a test process is sending its results. The operational state is enabled.

-terminating: The resource is in a termination phase. If the resource does not reinitialize itself autonomously, the Initialization Required condition is also present and the operational state is disabled. Otherwise, the operational state may be either disabled or enabled, depending upon the managed object class definition.

Public Attributes:

INIT_REQUIRED : int = 0x00000020

NOT_INITIALIZED : int = 0x00000040

INITIALIZING : int = 0x00000080

REPORTING : int = 0x00000100

TERMINATING : int = 0x00000200

PROCEDURAL_TYPE : short = 16

Public Operations:

getProceduralStatus () : int

Called to get the current set of values of the Procedural Status.

6.2.6 UsageState

This interface defines the Mask for Usage State.

-Idle: The resource is not currently in use.

-Active: The resource is in use, but has spare operating capacity to provide additional users at this instant.

-Busy: The resource is in use, but has no spare operating capacity to provide additional users at this instant.

Public Attributes:

IDLE : int = 0x00000020

ACTIVE : int = 0x00000040

BUSY : int = 0x00000080

USAGE_TYPE : short = 4

Public Operations:

getUsageState () : int

Called to get the current value of the Usage State.

6.2.7 ObjectStates

This interface allows objects which are meant to be managed in a standard way to implement a unified interface that supports all, or a suitable subset of, states and status values. The defined state and status attributes are specified by the ITU-T standard X.731 for State Management.

Derived from AlarmStatus, ProceduralStatus, AvailabilityStatus, UsageState, OperationalState, and AdministrativeState.

Public Operations:

getStatesSupported () : short[]

Called to determine which state and status attributes are supported by the class implementing this interface.

addStateChangeListener (listener : StateChangeListener):

Called to register a StateChangeListener for StateChangeEvents.

removeStateChangeListener (listener : StateChangeListener) :

Called to deregister a StateChangeListener.

getCurrentState () : int

Called to get the current value of all supported states. Returns a bit mask representing the individual states.

getCurrentStatus () : int

Called to get the current value of all supported status attributes. Returns a bit mask representing the individual status attributes.

6.2.8 StateChangeListener

This interface must be implemented by classes interested in being notified of state changes of objects that implement the GenericStates interface. If an object that is a StateChangeListener registers via the addStateChangeListener method, it will be notified by calling the stateChanged method, which includes the appropriate StateChangeEvent.

Public Operations:

stateChange (event : StateChangeEvent) :

Called to notify a StateChangeListener about a state change. The event parameter provides information about what state has changed.

6.2.9 ResourceStateException

A base Exception class related to the GenericState interface. This exception or its extensions are thrown when a invalid state change would be caused by a method call. Example: an object in a Disabled state cannot perform a certain operation unless it is Unlocked.

The class is derived from Exception.

Public Operations:

getState () : short

Called to determine which state consistency has been violated.

getValue () : int

Called to get the current value of the violated state.

6.2.10 StateChangeEvent

This Event is fired when a state changes its value. It is distributed to all registered StateChangeListeners. The event is derived from EventObject.

Public Operations:

getState () : short

Called to determine which state has changed.

getOldValue () : int

Called to determine the original value of the state.

getNewValue () : int

Called to determine the new value of the state.

getSourceIndicator () : short

Called to determine the cause of the event.

6.2.11 SourceIndicator

Public Attributes:

INTERNAL_CAUSE : short = 1

State change caused by an internal activity.

EXTERNAL_CAUSE : short = 2

State change caused by an external activity.

6.3 org.atsc.utility

This package provides a set of supporting and utility classes and interfaces used by other packages.

6.3.1 Registry

This interface provides a common root to all specialized registry interfaces, such as ApplicationRegistry, ResourceRegistry, etc. It is provided so that the RegistryFactory can return a base type. The interface is derived from RegistryType.

Public Operations:

getRegistryType () : String

Called to determine the type of registry implemented by the object returned by the RegistryFatory's method getRegistry().

6.3.2 RegistryFactory

This class provides a mechanism to create objects that implement specific Registry interfaces, such as the ApplicationRegistry.

Public Operations:

RegistryFactory () :

Constructor

getRegistry (registryName : String) : Registry

Returns an instance of an object that implements the specified registry interface. Returns null when specified registry does not exist or cannot be created. The type of the returned object will be one of the derived Registry types, such as the ApplicationRegistry.

6.3.3 RegistryType

This interface defines names for different registry types, such as an application registry, etc.

Public Attributes:

APPLICATION_REGISTRY : String = ApplicationRegistry

RESOURCE_REGISTRY : String = ResourceRegistry

Totals:

3 Logical Packages

23 Classes

Logical Package Structure:

```
Logical View
     java
          lang
          util
org
     atsc
          application
          management
          utility
     davic
          net
```

Accordingly, it can be seen that the present invention provides a software architecture for managing applications at a television set-top terminal. Application data, such as a program guide, stock ticker or the like, is recovered at the terminal according to a locator associated with the application data. The application data is registered and installed at the terminal, and a user is notified of the presence of the application data after registration and installation thereof.

Although the invention has been described in connection with various specific embodiments, those skilled in the art will appreciate that numerous adaptations and modifications may be made thereto without departing from the spirit and scope of the invention as set forth in the claims.

For example, while various syntax elements have been discussed herein, note that they are examples only, and any syntax may be used.

Moreover, the invention is suitable for use with virtually any type of network, including cable or satellite television broadband communication networks, local area networks (LANs), metropolitan area networks (MANs), wide area networks (WANs), internets, intranets, and the Internet, or combinations thereof.

Additionally, known computer hardware, firmware and/or software may be used to implement the invention.